Computing the Gale-Shapley Algorithm in R: Performance

Jan Tilly

May 17, 2015

In this vignette, I explore the computational performance of the Gale-Shapley algorithm (Gale and Shapley 1962) implemented in this package. The computational performance highly depends on how similar agents' preferences are. When some woman is the most preferred to many men, the algorithm will require a lot more rounds to compute the stable matching than when preferences are completely random. To capture this I will construct preferences that feature a common and an idiosyncratic component. The weight on the common component ("the level of commonality") is denoted by λ. When $\lambda = 1$, all men have identical preferences over all women and vice versa. When $\lambda = 0$, all preferences are completely random.

Preferences for men are then constructed as follows:

```
# number of men and women, respectively
n = 100
# level of commonality
lambda = 0.5
# men's preferences
uM = lambda * matrix(runif(n), nrow = n, ncol = n) +
          (1 - lambda) * runif(n ^ 2)
```

The common component `matrix(runif(n), nrow = n, ncol = n)` is a matrix of dimension `n` by `n` that has identical columns. The idiosyncratic component `runif(n ^ 2)` is a matrix of dimension `n` by `n` where each element is a separate draw from a uniform distribution.

Women's preferences are constructed similarly:

```
# womens's preferences
uW = lambda * matrix(runif(n), nrow = n, ncol = n) +
          (1 - lambda) * runif(n ^ 2)
```

I benchmark the matching functions for six different market sizes

```
# market sizes
N = c(100, 200, 250, 500, 1000)
```

and five different levels of commonality

```
# levels of commonality
Commonality = c(0.0, 0.25, 0.5, 0.75, 0.99)
```

One-to-one matching

The test function for the case of `one2one` matching is constructed as follows:

```
set.seed(1)

test_one2one = function(n, lambda) {
    uM = lambda * matrix(runif(n), nrow = n, ncol = n) +
         (1 - lambda) * runif(n ^ 2)
    uW = lambda * matrix(runif(n), nrow = n, ncol = n) +
         (1 - lambda) * runif(n ^ 2)
    one2one(uM, uW)
}
```

In this example, there are equal numbers of men and women, so everyone will get matched. Note that I am benchmarking the generation of preferences and the computation of the stable matching jointly.

Table 1: Run time (in seconds) for the matching of men to women

	lambda=0.00	lambda=0.25	lambda=0.50	lambda=0.75	lambda=0.99
N=100	0.002454	0.002602	0.002675	0.009082	0.002896
N=200	0.010147	0.009637	0.009925	0.010339	0.012930
N=250	0.019504	0.021870	0.016746	0.016295	0.021491
N=500	0.063363	0.071875	0.076679	0.078249	0.104776
N=1,000	0.298295	0.336828	0.350863	0.384550	0.853446

One-to-many matching

The test function for the case of `one2many` matching is constructed as follows:

```
set.seed(1)

test_one2many = function(n, lambda) {
    uWorkers = lambda * matrix(runif(n), nrow = n, ncol = n/10) +
          (1 - lambda) * runif(n ^ 2 / 10)
    uFirms = lambda * matrix(runif(n/10), nrow = n/10, ncol = n) +
          (1 - lambda) * runif(n ^ 2 / 10)
    one2many(uWorkers, uFirms, slots = 10)
}
```

In this example, I am matching workers with multi-worker firms, where each firm has 10 vacant positions. There are ten times as many workers as firms so every worker will get matched and every vacancy will be filled. Again, note that I am benchmarking the generation of preferences and the computation of the stable matching jointly.

Table 2: Run time (in seconds) for the matching of workers to multi-worker firms.

	lambda=0.00	lambda=0.25	lambda=0.50	lambda=0.75	lambda=0.99
N=100	0.001615	0.001866	0.001814	0.001830	0.002141
N=200	0.006692	0.013010	0.006466	0.006433	0.006392
N=250	0.008588	0.010200	0.010021	0.011093	0.017538
N=500	0.041897	0.053354	0.048577	0.055112	0.043794
N=1,000	0.187179	0.221889	0.250965	0.240545	0.247324

Literature

Gale, David, and Lloyd S Shapley. 1962. "College Admissions and the Stability of Marriage." *American Mathematical Monthly*, 9–15.